s1770036
Ping Goh

ILP report

Contents

s1770036
Ping Goh

Software architecture description

Architectural Goals and Constraints

Given a PowerGrab map with pre-existing stations, a drone will have to find a path that collects as many coins from as many positive stations as possible. The **stateless** drone has a limited look-ahead constraint (a 0.0003degrees radius) and cannot decide which direction to take if it doesn't sense any stations in its vicinity – the drone resorts to taking a random direction. The **stateful** drone can observe the whole map to choose the closest positive station and head towards it. Both drones aim to avoid negative-charged stations.

Classes Brief Overview

App
- User input states
  - 1) a date, of which the PowerGrab map will be retrieved from
  - 2) the drone's initial position on the map
  - 3) a seed to initialize the pseudo-random number generator
  - 4) a choice of either 'stateless' or 'stateful' drone
- Initializes a drone of choice and implement the appropriate game play algorithm on the provided map.

Station
- Template of object instances of type 'Stations', which are collected from a PowerGrab map's set of features.
- Only a station's coins and power values can be manipulated. Its position is fixed on its map.

- **Drone**
  - An abstract class Drone', a super class of classes 'Stateful' and 'Stateless'.
  - Allows manipulation of an abstract drone's properties: coins, power, position, and moves.

- **Stateless**
  - Template of object instances of type 'Stateless', a subclass of 'Drone', and inherits all the properties a drone has.
  - Differentiates from the other drone type 'Stateful', by implementing a more naïve algorithm of visiting positive stations and avoiding negative stations, with a one-step limited-lookahead constraint.

- **Stateful**
  - Template of object instances of type 'Stateful', a subclass of 'Drone', and inherits all the properties a drone has.
  - Differentiates from the other drone type Stateless, by implementing a Greedy algorithm of approaching to the next closest station to drone's current position. Achieved without the limited-lookahead constraint and determines the next closest station by looking at the whole map.

- **Position**
  - Template of object instances of type 'Position' – a property of a drone and station that keeps track of positions of drone and stations.
  - Identify if the drone is in which stations' vicinity by computing distances between drone and a station.

- **Direction**
  - Template of object instances of type 'Direction'. Stores the 16 directions a drone can take at each move.

s1770036
Ping Goh

Class documentation

Class	Methods/Attributes
Station	**Attributes**: ▪ **Float** *coin* ▪ **Float** *power* ▪ **Double** *p_coord* ▪ **Position** *position* The *coin* and the *power* attributes represent the drone's coins and power values, and are manipulated in the game (either increases or decreases depending which station it passes). The attribute *position* of a drone is mainly used to determine a drone's location on the map – which is created based on coordinate (latitude and longitude) values of the *p_coord* attribute.
	Methods: 1) **drone_visited()** - Use: Invoked when a drone enters this station's vicinity, hence its power and coin values are depleted by the drone. - Input: Called with a station object. - Effect: Updating this station's coin and power values to 0. - Output: *void method*
	2) **remove_from_stations()** - **Use**: During Stateful drone gameplay, where the drone can remember which positive stations it has visited, and it will only aim for the untouched positive stations on the map. Hence, the list of positive stations available for visit is reduced by one station (the one the drone just encountered). - **Input**: Called with a station object and input a list of stations as parameters. - **Effect:** Removes this station object from the list of stations. - **Output**: Returns the remaining list of stations.

s1770036
Ping Goh

<table>
<tr><td rowspan="3">Drone</td><td><u>Attributes:</u>
▪ Position position
▪ Int moves
▪ Float power
▪ Float coin
▪ Point point
▪ List<Point> drone_path
Drone_path is a list of points, denoting a drone's path so far. For stateless drone, this list is only used to compute the LineString paths on GeoJson map. For the stateful drone, the drone can look back points it has visited to decide its next move.</td></tr>
<tr><td><u>Methods:</u>
Constructor:
- When a stateless/stateful drone is initialized, the number of moves, power and coin values start with 250, 250 and 0, respectively. Its initial position (latitude and longitude) is set by user's input.

1) add_to_path()
- <u>Use</u>:
- Records all the points drone has travelled so far, for GeoJson's feature to compute a path line (LineString) between a pair of points.
- <u>Input</u>:
Called with a drone object.
- <u>Effect</u>:
Appends the drone's current position to the drone's existing list of points (drone_path) it has travelled.
- <u>Output</u>:
void method</td></tr>
<tr><td>2) one_move(Direction direction)
- <u>Use</u>:
Checks if drone has successfully moved. If it has, its number of moves and power value depletes and its point is added to its drone_path. If the drone's 250 moves hasn't run out, but the drone's power ran out first, drone cease to take the next step.
- <u>Input</u>:
Called with a drone object, along with a direction as input parameters.
- <u>Effect</u>:
This method updates the drone's position based on the input direction[line 43], converts drone's current position to point type and add it to drone_path.
- <u>Output</u>:
Returns Boolean true if drone successfully updates all its attributes after stepping into one direction. Returns Boolean false if drone's power is not sufficient to make one more move.</td></tr>
</table>

s1770036
Ping Goh

Stateless	1) **calculate_future_pos(List<Direction>** directions**)** - **Use**: For the drone to know its 16 next possible positions, from its current position. - **Input**: List of 16 directions which the drone can move in. - **Effect**: Iterates across each of the 16 possible directions to compute the corresponding potential position. - **Output**: Returns a list of positions the drone can head to in one move.
	2) **get_best_station(List<Station>** nearby_stations, **Random** rnd**)** - **Use**: After drone detects it has tapped into several stations' vicinity with a radius of 0.00025 degrees (stored as a list of *nearby_stations*), it finds the station with the most coin value and returns its index in the list of *nearby_stations*. - **Input**: list of *nearby_stations* drone has detected (by the **position_near_station** method in class Position). - **Effect**: Iterate across all stations in the given *nearby_stations* list, compares and remembers the station with the highest coin value at that iteration. Then, stores the index of this station with respect to the list - **Output**: Returns an integer, representing the interested index of the list.
	3) **to_random_dir(Random** rnd, **List<Station>** stations**)** - **Use**: In two different cases – 1) when a drone is about to enter a bad station's vicinity, it heads to a random direction as long it doesn't tap into another bad station's vicinity; 2) when no stations are in drone's vicinity, a drone has to make a random choice of direction to progress in the game, while avoiding bad stations. - **Input**: A random generator and a list of all stations in the map. - **Effect**: Obtain a random direction (*rand_dir*) using the random generator (*rnd*) parameter on the fixed set of 16 directions (*directions*). As long the next position predicted by the *rand_dir* variable results 1) outside of PlayArea or 2) encounters a bad station, then keep generating another new random direction. - **Output**: Returns a random direction generated, that results in a suitable next position.

	4) **to_station**(**Station** station, **Direction** direction, **Random** rnd, **List<Station>** stations) - **Use**: After identifying the direction that the drone should take to head to its goal positive station, it makes a move. But if it will also step into a bad station's vicinity, it chooses another random direction. - **Input**: 1) (*station*) the highest value station in the drone's vicinity, 2) (*direction*) the direction that leads to this goal station, 3) (*rnd*) the random generator, 4) (*stations*) a list of stations. - **Effect**: The drone proceeds to that said direction by invoking the class Drone's **one_move()** method. At the same time, the drone's power and coin values are updated by the encountered station's assets, and hence depleting the stations assets (by the **drone_visited()** method). But if drone is about to step into this positive station's vicinity and also steps into an overlapping bad station, then drone needs to take a random direction until it doesn't tap into the bad station's vicinity. - **Output**: Confirms and returns the direction drone has taken.
	5) **StartGameStateless(List<Station>** station_list, **Random** rnd**)** - **Use**: Implements a simple algorithm for the stateless drone gameplay, and returns the drone's path after its random search for positive stations to visit. - **Input**: A list of all stations collected from the GeoJson map and the random generator. - **Effect**: First, a list of nearby stations is collected for a drone's predicted next position. From this list of nearby stations, find the highest valued station to head to. If the list of nearby stations are all negative stations, pick another random direction that won't lead into them. Or if there are no nearby stations, continue to pick a random direction to head to. - **Output**: Return the drone's path (*drone_path*) in the form of GeoJson's LineString.

s1770036
Ping Goh

<table>
<tr><td rowspan="3">Stateful</td><td>1) closest_station(List<Station> stations)
- <u>Use</u>:
In the stateful drone strategy, drone looks at all the stations existing on the map and finds the closest one to make its way towards it.
- <u>Input</u>:
A list of all stations on the map.
- <u>Effect</u>:
This method iterates through all the stations in the list stations and compute the distance between each station and the drone’s current position.
- <u>Output</u>:
Returns the single station that is closest to the drone.</td></tr>
<tr><td>2) choose_direction(Station goal_station, List<Direction> directions)
- <u>Use</u>:
By knowing the closest station (goal_station), drone starts to make its way to it by identifying the best direction to get there.
- <u>Input</u>:
the closest station to the drone and a list of directions the drone can move with.
- <u>Effect</u>:
Iterate through the directions options to find the one that results the drone to end up nearest to the goal station. The chosen direction should not only lead drone closest to the goal, but also not step into a bad station area. If it does, choose the next best alternative direction.
- <u>Output</u>:
Returns the best direction.</td></tr>
<tr><td>3) met_station(Station station, Position pos)
- <u>Use</u>:
Checks if the drone’s next position (pos) will enter a specific station’s vicinity.
- <u>Input</u>:
A station and the drone’s predicted next position.
- <u>Effect</u>:
This method is quite straight forward and uses the class Position’s calculate_distance to compute distance between the drone’s next position and the station’s position.
- <u>Output</u>:
Returns a Boolean true if drone’s next position taps into a station’s vicinity. And a Boolean false, otherwise.</td></tr>
</table>

4) **met_bad_station(List<Station>** neg_stations, **Position** pos**)**

- **Use**:
  Checks if the drone's next position (*pos*) will enter any bad station's vicinity.
- **Input:**
  A list of negative stations and the drone's predicted next position.
- **Effect**:
  This method uses the **met_station** method to check if at least one bad station is met.
- **Output**:
  Returns a Boolean true of drone's next position taps into a bad station's vicinity. And false, otherwise.

5) **to_goal_station(List<Station>** pos_stations, **List<Station>** neg_stations**)**

- **Use**:
  Each time this method is invoked, drone incrementally moves the direction towards the goal station (*goal_station*).
- **Input:**
  List of positive stations (to identify the closest/goal station) and a list of negative stations to avoid them.
- **Effect**:
  For a successful move, if drone reaches the goal station, then drone's assets are updated accordingly. But if the goal station is overlapped by a bad station, or if goal station hasn't been reached but a bad station is encountered along the way, then drone need to take an alternative direction.
- **Output**:
  Returns the best direction that routes to the goal station, and avoids bad stations.

6) **get_all_pos_stations(List<Station>** stations**)**

- **Use**:
  Retrieve a list of positive stations for drone to focus on to visit.
- **Input:**
  List of all stations on the map.
- **Effect**:
  identify and store stations with coin values more than 0.
- **Output**:
  Return a list of positive stations.

7) **get_all_neg_stations(List<Station>** stations**)**

- **Use**:
  Retrieve a list of positive stations for drone to avoid.
- **Input:**
  List of all stations on the map.
- **Effect**:
  identify and store stations with coin values less than 0.
- **Output**:
  Return a list of positive stations.

	8) **StartGameStateful(List<Station>** station_list, **Random** rnd**)** - **Use**: Implements a greedy search algorithm for the stateful drone gameplay, and returns the drone's path after its calculated search for closest positive stations to visit. - **Input:** A list of all stations on the map, and a random generator. - **Effect**: There are some cases where drone can get stuck in a looped path. Most likely because there's no way to reach a particular goal positive station while avoiding the overlapped bad station. Hence, to break this loop, a condition is applied to remove this goal station from the list of positive stations to visit – which allows the drone to forget about this positive station and head to the next closest positive station. - **Output**: Return the drone's path (*drone_path*) in the form of GeoJson's LineString.
Position	1) **nextPostition(Direction** direction**)** - **Use**: Predicts the drone's next position. Not yet actually updating its position and appending it to its path. - **Input:** Called with the drone's position, and input direction that results its next position. - **Effect**: considers cases of each 16 directions to predict the new drone position, using the Euclidean distance to compute the distances between two positions. - **Output**: Return the (predicted) position.
	2) **inPlayArea()** - **Use**: checks if the drone's position is in the valid play area. - **Input:** method called with the object position. - **Effect**: comparing the latitudes and longitudes of a drone against the fixed latitudes and longitudes of the play area. - **Output**: Returns Boolean true if drone is in play area. False, if otherwise.

	3) **position_near_station(Station** station**)** - **Use**: checks if there's a station nearby a particular position. - **Input:** Called with a position along with an input station. - **Effect**: by the Euclidean distance between these two positions, if they are less than 0.00025. - **Output**: Returns Boolean true if a station is nearby, and false otherwise.
	4) **bad_stations_near(List<Station>** stations**)** - **Use**: checks if there's any bad nearby station. Drone can know not to go to next position if it will end up encountering the bad station. - **Input:** Called with the drone's position, along with a list of all stations on map. - **Effect**: Checks each station on the map, if it's a bad station and the drone will encounter it, return Boolean true. - **Output**: Returns Boolean true if at least one bad station is nearby, and false otherwise.
	5) **calculate_distance(Position** p2**)** - **Use**: computes distance between two positions. - **Input:** called with a position object, along with an input parameter of another position (*p2*). - **Effect**: Using the Euclidean distance measure. - **Output**: Return the distance of type double.

	6) **head_random(Random** rnd, **List<Station>** neg_stations**)** - **Use**: heads to random direction, while avoiding negative stations. - **Input:** random generator to compute random directions, and list of negative stations to avoid. - **Effect**: Generates a random direction. Checks if it leads drone to undesirable destinations (e.g. our of play area, and into bad stations). If it does, keep generating a new random direction until results a desirable destination. - **Output**: Returns the direction that doesn't lead to our of play area nor enter a bad station's vicinity.
Direction	1) **remove_from_directions(List<Direction>** directions**)** - **Use**: when a drone has tried a direction and failed to get into a suitable position, then remove this direction option from its list of directions. - **Input:** Called with a direction object, along with a list of directions as input parameters. - **Effect**: creates a new list to store all directions except the direction object called with. Hence, resulting a new list with one less direction. - **Output**: a list of new directions. 2) **directions()** - **Use**: Retrieves a list of 16 directions drone can move in. - **Input:** Called without instantiating a direction object. - **Output**: return a list of 16 directions.

s1770036
Ping Goh

Stateful drone strategy

The stateful drone remembers the positive stations it has visited and doesn't visit them again. This is achieved by initially storing all the positive stations on the map, and slowly removing each station from the list as the drone visits them. This allows the drone to not make unnecessary calculations to travel back to stations it has visited before. Unlike the stateless drone, where its random paths can be observed travelling back to a green station it already has visited, as observed in Fig.2's center.

The stateful drone also remembers the directions it has tried that failed, so it wouldn't head to the same unfavorable spot again. E.g. it knows that by heading North, it will enter a bad station's vicinity, so it wont

Another point that makes the stateful drone perform better is that it gets to oversee the entire map before it makes each move. Unlike the stateless drone which has a constraint of only 1 step look-ahead. This ability also enables the stateful drone to not need to depend on heading to random directions when deciding where to go next. Because it knows where its targeted station is, it knows the best direction that allows the drone to end up closer to its target. This can be clearly seen in Fig.1 for the clean neat single lines where the drone knows which station to head to.

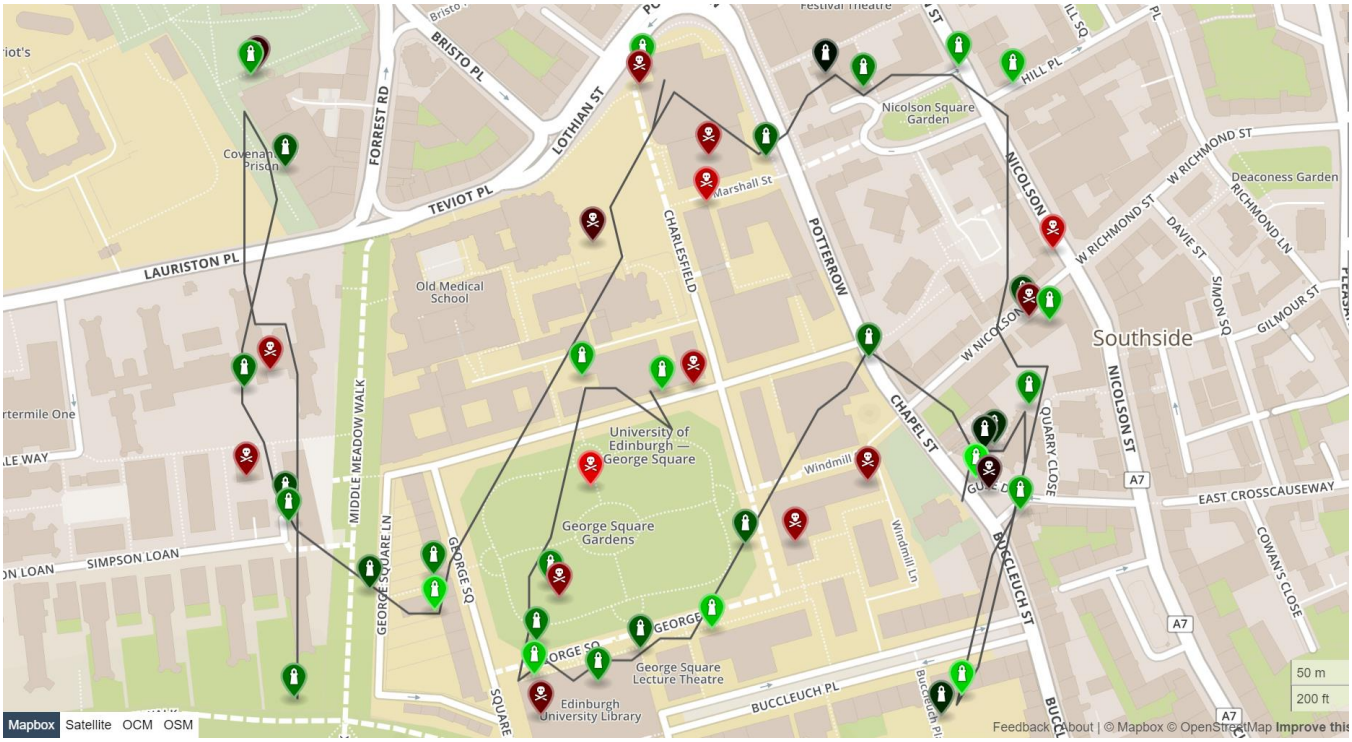

Figure 1: Stateful drone path

s1770036
Ping Goh

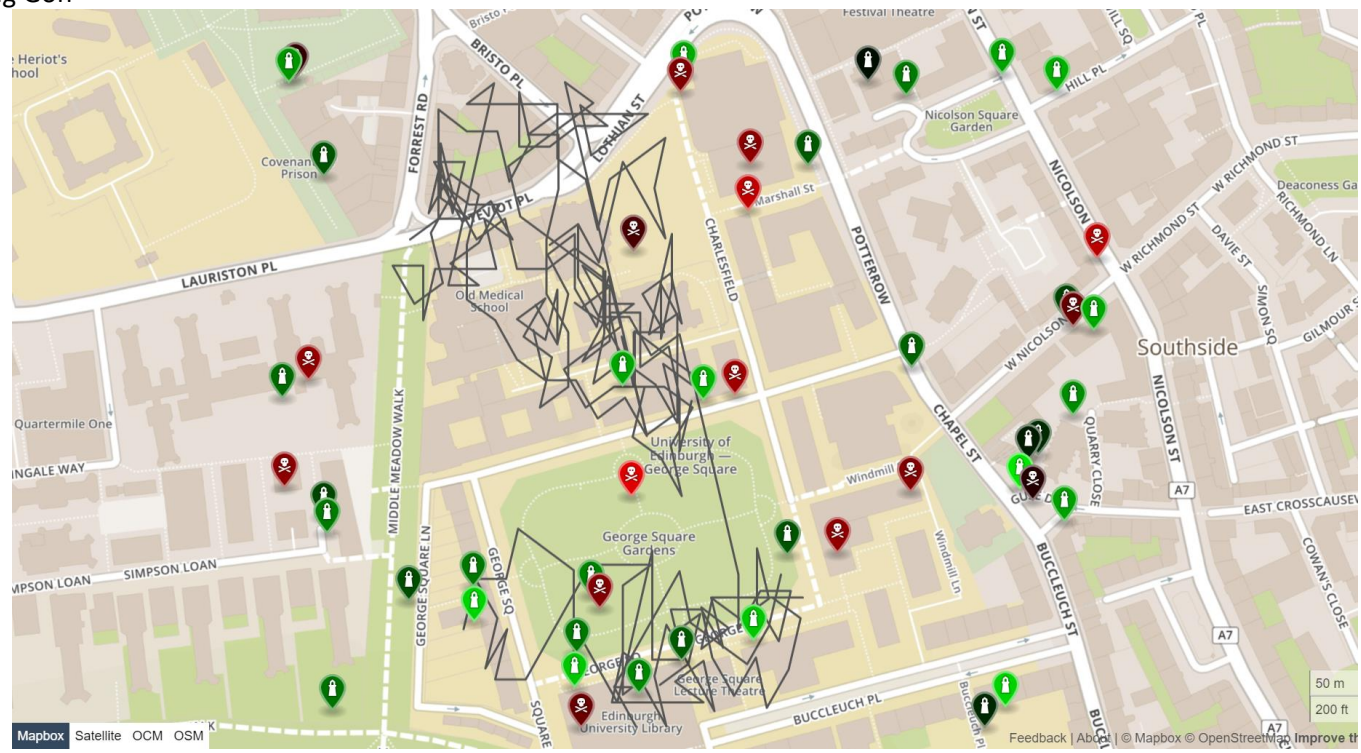

Figure 2: Stateless drone path